Outline

- Graph generalities
- Graph processing
- Storing the graph
- Aside: Variable-length representations
- Graph compression with instantaneous codes: BVGraph
- BVGraph for general graphs: LLPA
- Graph compression with Elias-Fano: EFGraph

Graph generalities

Graph

A graph $G = (V_G, E_G)$ is defined by:

Graph

A graph $G = (V_G, E_G)$ is defined by:

Graph

A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of *nodes*

Graph

A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of *nodes*
- A set $E = E_G \subseteq V \times V$ of *arcs* (ordered pairs of nodes).

Graph

A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of *nodes*
- A set $E = E_G \subseteq V \times V$ of *arcs* (ordered pairs of nodes).

Other authors call these *directed graphs* (or digraphs, or networks).

Graph

A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of *nodes*
- A set $E = E_G \subseteq V \times V$ of *arcs* (ordered pairs of nodes).

Other authors call these *directed graphs* (or digraphs, or networks). Note that pairs of the form (x, x) (*loops*) are allowed (many authors don't take loop into consideration).

Graph

A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of *nodes*
- A set $E = E_G \subseteq V \times V$ of *arcs* (ordered pairs of nodes).

Other authors call these *directed graphs* (or digraphs, or networks). Note that pairs of the form (x, x) (*loops*) are allowed (many authors don't take loop into consideration). G is often dropped from the indices, when clear from the context.

Transpose graph

The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$

Transpose graph

The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$

A graph G is *symmetric* iff $G = G^T$.

Transpose graph

The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$

A graph G is *symmetric* iff $G = G^T$.

We similarly define $G^s = (V, E \cup E^T)$ (the symmetric closure of G).

Transpose graph

The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$

A graph G is *symmetric* iff $G = G^T$.

We similarly define $G^s = (V, E \cup E^T)$ (the symmetric closure of G).

Transpose graph

The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$

A graph G is *symmetric* iff $G = G^T$.

We similarly define $G^s = (V, E \cup E^T)$ (the symmetric closure of G).

- *Undirected graphs* can be safely identified with symmetric graphs.

Transpose graph

The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$

A graph G is *symmetric* iff $G = G^T$.
We similarly define $G^s = (V, E \cup E^T)$ (the symmetric closure of G).

- *Undirected graphs* can be safely identified with symmetric graphs.
- In an undirected graphs, nodes are often called *vertices* and pairs of opposite arcs are called *edges*.

Multigraphs and hypergraphs

In some applications, one may want more than one arc between two nodes (i.e., that E is a multiset of pairs, instead of a set). We call these generalization *multigraphs*.

Multigraphs and hypergraphs

In some applications, one may want more than one arc between two nodes (i.e., that E is a multiset of pairs, instead of a set). We call these generalization *multigraphs*.

In some other applications, E is not a set of pairs, but a set of r-tuples. In this case, we talk of *hypergraphs*.

Labels

A graph can be labelled on its nodes and/or on its arcs.

Node-labelling functions map nodes (or arcs) to a set of suitable labels.

Labels

A graph can be labelled on its nodes and/or on its arcs.
Node-labelling functions map nodes (or arcs) to a set of suitable labels.
In the case of undirected graphs one usually requires that (x, y) has the same label as (y, x), so that one can think of labels being assigned to edges.

Labels

A graph can be labelled on its nodes and/or on its arcs.
Node-labelling functions map nodes (or arcs) to a set of suitable labels.
In the case of undirected graphs one usually requires that (x, y) has the same label as (y, x), so that one can think of labels being assigned to edges.
A special case of labelling is the assingnment of real values, that is often called a *weighting function* (hence we call a graph node-weighted or arc-weighted).

Graphs everywhere. . .

Graphs (of some kind) pop up everywhere. Examples:

Graphs everywhere...

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)

Graphs everywhere...

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)

Graphs everywhere...

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs

Graphs everywhere. . .

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems

Graphs everywhere. . .

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems
- semantic networks, knowledge graphs

Graphs everywhere...

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems
- semantic networks, knowledge graphs
- collaboration graphs

Graphs everywhere. . .

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems
- semantic networks, knowledge graphs
- collaboration graphs
- citation networks

Graphs everywhere. . .

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems
- semantic networks, knowledge graphs
- collaboration graphs
- citation networks
- . . .

Sizes

These graphs may have a *humongous* number of vertices (not rarely, they have billions of nodes!).

Sizes

These graphs may have a *humongous* number of vertices (not rarely, they have billions of nodes!).
Typically, though, they are very *sparse*:

Sizes

These graphs may have a *humongous* number of vertices (not rarely, they have billions of nodes!).
Typically, though, they are very *sparse*: A *sparse graph* is one with $O(n)$ arcs (instead of $O(n^2)$).

Paths

A *path* in G is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k-1$. We say that:

Paths

A *path* in G is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k-1$. We say that:

Paths

A *path* in G is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k-1$. We say that:

- π starts at node x_0 (also called the *source* of π)

Paths

A *path* in G is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k-1$. We say that:

- π starts at node x_0 (also called the *source* of π)
- π ends at node x_k (also called the *target* of π)

Paths

A *path* in G is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k-1$. We say that:

- π starts at node x_0 (also called the *source* of π)
- π ends at node x_k (also called the *target* of π)
- has *length* $|\pi| = k$

Paths

A *path* in G is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k-1$. We say that:

- π starts at node x_0 (also called the *source* of π)
- π ends at node x_k (also called the *target* of π)
- has *length* $|\pi| = k$
- is *simple* if $x_0, \ldots, x_{k-1}$ are all distinct

Paths

A *path* in G is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k-1$. We say that:

- π starts at node x_0 (also called the *source* of π)
- π ends at node x_k (also called the *target* of π)
- has *length* $|\pi| = k$
- is *simple* if $x_0, \ldots, x_{k-1}$ are all distinct
- it is a *cycle* iff $k > 0$ and the source and target coincide.

Paths

A *path* in G is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k-1$. We say that:

- π starts at node x_0 (also called the *source* of π)
- π ends at node x_k (also called the *target* of π)
- has *length* $|\pi| = k$
- is *simple* if $x_0, \ldots, x_{k-1}$ are all distinct
- it is a *cycle* iff $k > 0$ and the source and target coincide.
- if there is a path from x to y we say that y is *reachable* from x

Paths

A *path* in G is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k-1$. We say that:

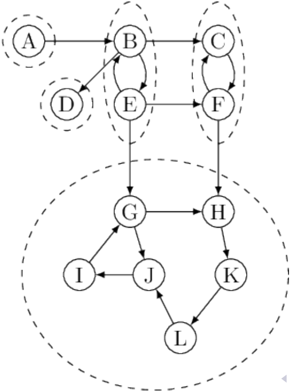
- π starts at node x_0 (also called the *source* of π)
- π ends at node x_k (also called the *target* of π)
- has *length* $|\pi| = k$
- is *simple* if $x_0, \ldots, x_{k-1}$ are all distinct
- it is a *cycle* iff $k > 0$ and the source and target coincide.
- if there is a path from x to y we say that y is *reachable* from x
- if there is a cycle, G is called *cyclic*.

Strongly connected components

Let $x \approx y$ iff there is a path from x to y and vice-versa. The equivalence classes of $\approx$ are called the *strongly connected components* (SCCs) of G. The SCCs of G^s are called the *weakly connected components* (WCCs) of G: in the case of a symmetric graph, WCCs and SCCs coincide (and we just talk of "connected components").

Strongly connected components

Let $x \approx y$ iff there is a path from x to y and vice-versa. The equivalence classes of $\approx$ are called the *strongly connected components* (SCCs) of G. The SCCs of G^s are called the *weakly connected components* (WCCs) of G: in the case of a symmetric graph, WCCs and SCCs coincide (and we just talk of "connected components").



Strongly connected components

The *reduced graph* $G^\dagger$ is the graph whose nodes are the SCCs of G, with an arc from $[x]$ to $[y]$ whenever there is a node $x' \approx x$ and a node $y' \approx y$ with $(x', y') \in E$.

Strongly connected components

The *reduced graph* $G^\dagger$ is the graph whose nodes are the SCCs of G, with an arc from $[x]$ to $[y]$ whenever there is a node $x' \approx x$ and a node $y' \approx y$ with $(x', y') \in E$.

Theorem

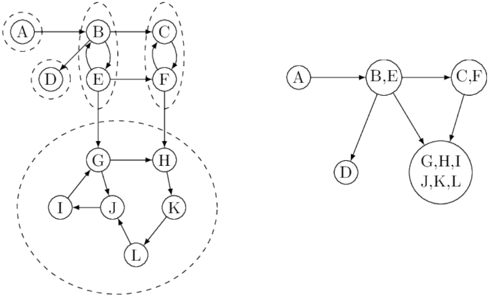
$G^\dagger$ is an acyclic graph.

Strongly connected components

The *reduced graph* $G^\dagger$ is the graph whose nodes are the SCCs of G, with an arc from $[x]$ to $[y]$ whenever there is a node $x' \approx x$ and a node $y' \approx y$ with $(x', y') \in E$.

Theorem

$G^\dagger$ is an acyclic graph.



Neighborhoods and degrees

Given $G = (V, E)$ and $x \in V$, define:

Neighborhoods and degrees

Given $G = (V, E)$ and $x \in V$, define:

- $N_G^-(x) = \{y \mid (y, x) \in E\}$ (in-neighborhood of x, predecessors of x)

Neighborhoods and degrees

Given $G = (V, E)$ and $x \in V$, define:

- $N_G^-(x) = \{y \mid (y, x) \in E\}$ (in-neighborhood of x, predecessors of x)
- $N_G^+(x) = \{y \mid (x, y) \in E\}$ (out-neighborhood of x, successors of x)

Neighborhoods and degrees

Given $G = (V, E)$ and $x \in V$, define:

- $N_G^-(x) = \{y \mid (y, x) \in E\}$ (in-neighborhood of x, predecessors of x)
- $N_G^+(x) = \{y \mid (x, y) \in E\}$ (out-neighborhood of x, successors of x)
- $d_G^-(x) = |N_G^-(x)|$ (in-degree of x)

Neighborhoods and degrees

Given $G = (V, E)$ and $x \in V$, define:

- $N_G^-(x) = \{y \mid (y, x) \in E\}$ (in-neighborhood of x, predecessors of x)
- $N_G^+(x) = \{y \mid (x, y) \in E\}$ (out-neighborhood of x, successors of x)
- $d_G^-(x) = |N_G^-(x)|$ (in-degree of x)
- $d_G^+(x) = |N_G^+(x)|$ (out-degree of x)

(Local) clustering coefficient

Given $G = (V, E)$ and $x \in V$, define:

(Local) clustering coefficient

Given $G = (V, E)$ and $x \in V$, define:

- in-directed clustering coefficient of x:

$$c_G^-(x) = \frac{|E_G \cap (N_G^-(x) \times N_G^-(x)|}{d_G^-(x)^2}$$

or, if loop are not allowed:

$$c_G^-(x) = \frac{|E_G \cap (N_G^-(x) \times N_G^-(x)|}{d_G^-(x) \cdot (d_G^-(x) - 1)}$$

(Local) clustering coefficient

Given $G = (V, E)$ and $x \in V$, define:

- in-directed clustering coefficient of x:

$$c_G^-(x) = \frac{|E_G \cap (N_G^-(x) \times N_G^-(x)|}{d_G^-(x)^2}$$

  or, if loop are not allowed:

$$c_G^-(x) = \frac{|E_G \cap (N_G^-(x) \times N_G^-(x)|}{d_G^-(x) \cdot (d_G^-(x) - 1)}$$

- $c_G^+(x)$ is defined similarly

(Local) clustering coefficient

Given $G = (V, E)$ and $x \in V$, define:

- in-directed clustering coefficient of x:

$$c_G^-(x) = \frac{|E_G \cap (N_G^-(x) \times N_G^-(x)|}{d_G^-(x)^2}$$

  or, if loop are not allowed:

$$c_G^-(x) = \frac{|E_G \cap (N_G^-(x) \times N_G^-(x)|}{d_G^-(x) \cdot (d_G^-(x) - 1)}$$

- $c_G^+(x)$ is defined similarly
- for undirected loopless graphs:

$$c_G(x) = \frac{2|E_G \cap (N_G(x) \times N_G(x)|}{d_G(x) \cdot (d_G(x) - 1)}$$

Graph morphism

A graph *morphism* $f : G \to H$ is a function $f : V_G \to V_H$ such that $(x, y) \in E_G$ if and only if $(f(x), f(y)) \in E_H$.

Graph morphism

A graph *morphism* $f : G \to H$ is a function $f : V_G \to V_H$ such that $(x, y) \in E_G$ if and only if $(f(x), f(y)) \in E_H$.
A bijective graph morphism is called an *isomorphism*. If there exists an isomorphism between G and H we say that G and H are isomorphic, and write $G \cong H$.

Graph processing

Graph property

A *graph property* is a function that associates a value to each graph.

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties
  - Average degree

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties
  - Average degree
  - Average (shortest-path) distance

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient
- Distributions

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient
- Distributions
  - Degree distribution

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient
- Distributions
  - Degree distribution
  - Distance distribution

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient
- Distributions
  - Degree distribution
  - Distance distribution
- Vector properties

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient
- Distributions
  - Degree distribution
  - Distance distribution
- Vector properties
  - Local clustering coefficient

Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?
- Scalar properties
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient
- Distributions
  - Degree distribution
  - Distance distribution
- Vector properties
  - Local clustering coefficient
  - Centrality (e.g., eccentricity)

Graph property

A graph property P is *isomorphism-invariant* iff

$$G \cong H \text{ implies } P(G) = P(H).$$

Graph property

A graph property P is *isomorphism-invariant* iff

$$G \cong H \text{ implies } P(G) = P(H).$$

Properties that are *not* isomorphism-invariant are tricky (they depend on the specific identity of nodes).

Graph property

A graph property P is *isomorphism-invariant* iff

$$G \cong H \text{ implies } P(G) = P(H).$$

Properties that are *not* isomorphism-invariant are tricky (they depend on the specific identity of nodes).
If we limit ourselves to isomorphism-invariant properties, we can assume w.l.o.g. that $V_G = \{0, 1, \dots, n-1\}$.

Accessing a graph

If you store a graph G (in some way), you can access it through different primitives, such as. . .

Accessing a graph

If you store a graph G (in some way), you can access it through different primitives, such as...

- direct access queries:

Accessing a graph

If you store a graph G (in some way), you can access it through different primitives, such as...

- direct access queries:
  - $G.\mathrm{arc}(x, y)$ (is there an arc from x to y in G?)

Accessing a graph

If you store a graph G (in some way), you can access it through different primitives, such as...

- direct access queries:
  - $G.\mathrm{arc}(x, y)$ (is there an arc from x to y in G?)
  - $G.\mathrm{d}^{\pm}(x)$ (what is the in/out-degree of x in G?)

Accessing a graph

If you store a graph G (in some way), you can access it through different primitives, such as...

- direct access queries:
  - $G.\mathrm{arc}(x, y)$ (is there an arc from x to y in G?)
  - $G.\mathrm{d}^{\pm}(x)$ (what is the in/out-degree of x in G?)
  - $G.\mathrm{N}^{\pm}(x)$ (enumerate the in/out-neighbors of x in G; possibly in order of id)

Accessing a graph

If you store a graph G (in some way), you can access it through different primitives, such as...

- direct access queries:
  - $G.\mathrm{arc}(x,y)$ (is there an arc from x to y in G?)
  - $G.\mathrm{d}^{\pm}(x)$ (what is the in/out-degree of x in G?)
  - $G.\mathrm{N}^{\pm}(x)$ (enumerate the in/out-neighbors of x in G; possibly in order of id)
- sequential access (a.k.a., streaming):

Accessing a graph

If you store a graph G (in some way), you can access it through different primitives, such as...

- direct access queries:
  - $G.\mathrm{arc}(x, y)$ (is there an arc from x to y in G?)
  - $G.\mathrm{d}^{\pm}(x)$ (what is the in/out-degree of x in G?)
  - $G.\mathrm{N}^{\pm}(x)$ (enumerate the in/out-neighbors of x in G; possibly in order of id)
- sequential access (a.k.a., streaming):
  - $G.E$ (enumerate the arcs, possibly preserving the consecutivity of in/out-neighborhoods)

Accessing a graph

If you store a graph G (in some way), you can access it through different primitives, such as...

- direct access queries:
  - $G.\mathrm{arc}(x, y)$ (is there an arc from x to y in G?)
  - $G.\mathrm{d}^{\pm}(x)$ (what is the in/out-degree of x in G?)
  - $G.\mathrm{N}^{\pm}(x)$ (enumerate the in/out-neighbors of x in G; possibly in order of id)
- sequential access (a.k.a., streaming):
  - $G.E$ (enumerate the arcs, possibly preserving the consecutivity of in/out-neighborhoods)
  - if $G.E$ can be called only once (or $O(1)$ times), the access is "streaming".

Computing graph properties

Typical problem:

Computing graph properties

Typical problem:

- for a given property P

Computing graph properties

Typical problem:

- for a given property P
- for a given access mode

Computing graph properties

Typical problem:

- for a given property P
- for a given access mode
- write an algorithm that computes (or approximates, in some sense) $G \mapsto P(G)$

Storing the graph

. . . "storing the graph"?

- Having a data structure that allows you, for a given node, to know its successors.

What do we mean by. . .

. . . "storing the graph"?

- Having a data structure that allows you, for a given node, to know its successors.
- If the graph is node-labelled (e.g., a web graph with URLs as node labels): having a way to know which label corresponds to a given node and vice versa.

What do we mean by...

... "storing the graph"?

- Having a data structure that allows you, for a given node, to know its successors.
- If the graph is node-labelled (e.g., a web graph with URLs as node labels): having a way to know which label corresponds to a given node and vice versa.

Here: we only consider the former problem, not the latter!

Information-theoretical lower bound

How much space do we need to store a graph with n nodes and m arcs?

Information-theoretical lower bound

How much space do we need to store a graph with n nodes and m arcs? Not less than

$$\log \binom{n^2}{m} \approx m \log \left(\frac{n^2}{m}\right) + O(m)$$

Information-theoretical lower bound

How much space do we need to store a graph with n nodes and m arcs? Not less than

$$\log \binom{n^2}{m} \approx m \log \left(\frac{n^2}{m} \right) + O(m)$$

under the hypothesis that $m = o(n^2)$

$$m \log \left(\frac{n}{d} \right) + O(m)$$

where $d = m/n$ is the average degree.

Information-theoretical lower bound

How much space do we need to store a graph with n nodes and m arcs? Not less than

$$\log \binom{n^2}{m} \approx m \log \left(\frac{n^2}{m} \right) + O(m)$$

under the hypothesis that $m = o(n^2)$

$$m \log \left(\frac{n}{d} \right) + O(m)$$

where $d = m/n$ is the average degree.
This means about $\log(n/d) + O(1)$ bits per arc. But *complex networks are NOT random graphs!*.

Naive graph representations (1)

Most popular naive representations for a graph G:

Naive graph representations (1)

Most popular naive representations for a graph G:

- adjacency matrix: a $n \times n$ binary matrix G with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Naive graph representations (1)

Most popular naive representations for a graph G:

- adjacency matrix: a $n \times n$ binary matrix G with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

Naive graph representations (1)

Most popular naive representations for a graph G:

- adjacency matrix: a $n \times n$ binary matrix G with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

- it occupies n^2 bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$)

Naive graph representations (1)

Most popular naive representations for a graph G:

- adjacency matrix: a $n \times n$ binary matrix G with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

- it occupies n^2 bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$)
- $G.\mathrm{arc}(x, y)$ takes constant time:

Naive graph representations (1)

Most popular naive representations for a graph G:

- adjacency matrix: a $n \times n$ binary matrix G with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

- it occupies n^2 bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$)
- $G.\mathrm{arc}(x, y)$ takes constant time:
- enumerating the neighborhoods takes time $O(n)$

Naive graph representations (1)

Most popular naive representations for a graph G:

- adjacency matrix: a $n \times n$ binary matrix G with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

- it occupies n^2 bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$)
- $G.\mathrm{arc}(x, y)$ takes constant time:
- enumerating the neighborhoods takes time $O(n)$
- scanning the graph sequentially takes time $O(n^2)$

Naive graph representations (1)

Most popular naive representations for a graph G:

- adjacency matrix: a $n \times n$ binary matrix G with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

- it occupies n^2 bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$)
- $G.\mathrm{arc}(x, y)$ takes constant time:
- enumerating the neighborhoods takes time $O(n)$
- scanning the graph sequentially takes time $O(n^2)$
- **highly unsuitable for sparse graphs!**

Naive graph representations (2)

Second most popular naive representations for a graph G:

Naive graph representations (2)

Second most popular naive representations for a graph G:

- adjacency lists: one list per node, containing its successors (in increasing order).

Naive graph representations (2)

Second most popular naive representations for a graph G:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:

Naive graph representations (2)

Second most popular naive representations for a graph G:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:

- memory occupied: see below

Naive graph representations (2)

Second most popular naive representations for a graph G:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:

- memory occupied: see below
- $G.\mathrm{a}rc(x, y)$ takes $O(d)$ time

Naive graph representations (2)

Second most popular naive representations for a graph G:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:

- memory occupied: see below
- $G.\mathrm{a}rc(x, y)$ takes $O(d)$ time
- enumerating the neighborhoods takes time $O(d)$

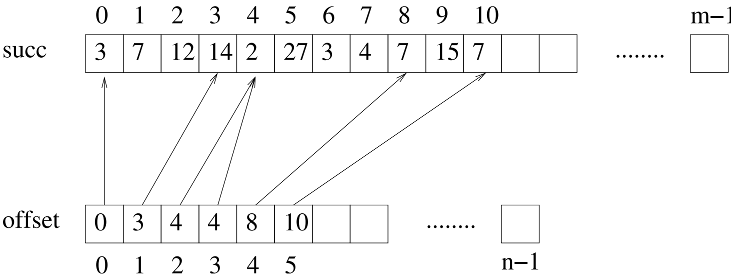
Naive graph representations (2)

Second most popular naive representations for a graph G:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:

- memory occupied: see below
- $G.\mathrm{arc}(x, y)$ takes $O(d)$ time
- enumerating the neighborhoods takes time $O(d)$
- scanning the graph sequentially takes time $O(m)$



The offset vector tells, for each given node x, where the successor list of x starts from. Implicitly, it also gives the degree of each node.

Naive representation

How much space does this representation take?

- Successor array: m elements (arcs), each containing a node ($\log n$ bits); with 32 bits, we can store up to 4 billion nodes (half of it, if we don't have unsigned types)

Naive representation

How much space does this representation take?

- Successor array: m elements (arcs), each containing a node ($\log n$ bits); with 32 bits, we can store up to 4 billion nodes (half of it, if we don't have unsigned types)
- Offset array: n elements (nodes), each containing an index in the successor array ($\log m$ bits); with 32 bits, we can store up t 4 billion arcs.

Naive representation

How much space does this representation take?

- Successor array: m elements (arcs), each containing a node ($\log n$ bits); with 32 bits, we can store up to 4 billion nodes (half of it, if we don't have unsigned types)
- Offset array: n elements (nodes), each containing an index in the successor array ($\log m$ bits); with 32 bits, we can store up t 4 billion arcs.

All in all, $32(n + m)$ bits. If we assume $m = 8n$ (a very modest assumption on the outdegree), we need $288n$ bits, i.e., 288 bits/node, 36 bits/arc.
We show how to reduce this of an order of magnitude.

Idea

Use a variable-length representation for successors. Such a representation should obviously...

- be instantaneously decodable

Idea

Use a variable-length representation for successors. Such a representation should obviously...

- be instantaneously decodable
- minimize the expected bitlength.

Use a variable-length representation for successors. Such a representation should obviously. . .

- be instantaneously decodable
- minimize the expected bitlength.

What about the offset array?

Idea

Use a variable-length representation for successors. Such a representation should obviously. . .

- be instantaneously decodable
- minimize the expected bitlength.

What about the offset array?

- bit displacement vs. byte displacement (with alignment)

Idea

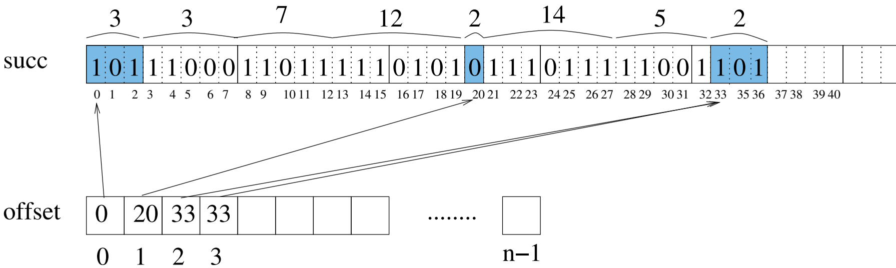
Use a variable-length representation for successors. Such a representation should obviously. . .

- be instantaneously decodable
- minimize the expected bitlength.

What about the offset array?

- bit displacement vs. byte displacement (with alignment)
- we have to keep an explicit representation of the node degrees (e.g., in the successor array, before every successsor list).

Variable-length representation



Node degrees (blue background), followed by successors. Each number is represented using an instantaneous code (possibly, different for degree and successors).

Aside: Variable-length representations

Instantaneous code

- An *instantaneous (binary) code* for the set S is a function $c : S \to \{0, 1\}^*$ such that, for all $x, y \in S$, if $c(x)$ is a prefix of $c(y)$, then $x = y$.

Instantaneous code

- An *instantaneous (binary) code* for the set S is a function $c : S \to \{0, 1\}^*$ such that, for all $x, y \in S$, if $c(x)$ is a prefix of $c(y)$, then $x = y$.
- Let l_x be the length (in bits) of $c(x)$.

Instantaneous code

- An *instantaneous (binary) code* for the set S is a function $c : S \to \{0,1\}^*$ such that, for all $x, y \in S$, if $c(x)$ is a prefix of $c(y)$, then $x = y$.
- Let l_x be the length (in bits) of $c(x)$.
- Kraft-McMillan: there exists an instantaneous code with lengths l_x ($x \in S$) if and only if

$$\sum_{x \in S} 2^{-l_x} \leq 1.$$

Instantaneous code

- An *instantaneous (binary) code* for the set S is a function $c : S \to \{0,1\}^*$ such that, for all $x, y \in S$, if $c(x)$ is a prefix of $c(y)$, then $x = y$.
- Let l_x be the length (in bits) of $c(x)$.
- Kraft-McMillan: there exists an instantaneous code with lengths l_x ($x \in S$) if and only if

$$\sum_{x \in S} 2^{-l_x} \leq 1.$$

- An instantaneous code for which the *equality* holds is called "complete".

Expected length

Given a set S and an instantaneous code $c : S \to \{0, \}^*$, the *expected length of c* with respect to some probability distribution $p : S \to [0, 1]$ is

$$E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.$$

Expected length

Given a set S and an instantaneous code $c : S \to \{0, \}^*$, the *expected length of* c with respect to some probability distribution $p : S \to [0, 1]$ is

$$E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.$$

- Given a probability distribution p, the *optimal code* is the instantaneous code $c_p^* : S \to \{0, 1\}$ minimizing $E_p[c_p^*]$.

Expected length

Given a set S and an instantaneous code $c : S \to \{0, \}^*$, the *expected length of c* with respect to some probability distribution $p : S \to [0, 1]$ is

$$E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.$$

- Given a probability distribution p, the *optimal code* is the instantaneous code $c_p^* : S \to \{0, 1\}$ minimizing $E_p[c_p^*]$.
- If S is finite:

Expected length

Given a set S and an instantaneous code $c : S \rightarrow \{0, \}^*$, the *expected length of c* with respect to some probability distribution $p : S \rightarrow [0, 1]$ is

$$E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.$$

- Given a probability distribution p, the *optimal code* is the instantaneous code $c_p^* : S \rightarrow \{0, 1\}$ minimizing $E_p[c_p^*]$.
- If S is finite:
  - $H[p] = E_p[c_p^*]$ (Shannon's coding theorem)

Expected length

Given a set S and an instantaneous code $c : S \to \{0, \}^*$, the *expected length of c* with respect to some probability distribution $p : S \to [0, 1]$ is

$$E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.$$

- Given a probability distribution p, the *optimal code* is the instantaneous code $c_p^* : S \to \{0, 1\}$ minimizing $E_p[c_p^*]$.
- If S is finite:
  - $H[p] = E_p[c_p^*]$ (Shannon's coding theorem)
  - c_p^* is the Huffman coding.

Intended distribution

Intended distribution

- Given an instantaneous code $c : S \to \{0,1\}^*$, define a $p : S \to [0,1]$ as
$$p(x) = 2^{-|c(x)|}.$$

Intended distribution

- Given an instantaneous code $c : S \to \{0,1\}^*$, define a $p : S \to [0,1]$ as
$$p(x) = 2^{-|c(x)|}.$$

- p is called the *intended distribution* for the code c.

Intended distribution

- Given an instantaneous code $c : S \to \{0,1\}^*$, define a $p : S \to [0,1]$ as
$$p(x) = 2^{-|c(x)|}.$$
- p is called the *intended distribution* for the code c.
- to be more precise: P is a probability distribution if c is complete (otherwise it does not sum up to 1, and we need to introduce some normalization factor to turn it into a distribution).

Intended distribution

- Given an instantaneous code $c : S \to \{0,1\}^*$, define a $p : S \to [0,1]$ as
$$p(x) = 2^{-|c(x)|}.$$
- p is called the *intended distribution* for the code c.
- to be more precise: P is a probability distribution if c is complete (otherwise it does not sum up to 1, and we need to introduce some normalization factor to turn it into a distribution).

It is easy to see that (if S is finite) c is the optimal code for p; in fact:

$$H(p) = \sum_{s \in S} -p(s) \log p(s) = \sum_{s \in S} 2^{-|c(s)|} |c(s)| = \sum_{s \in S} p(s)|c(s)| = E_p[c].$$

Intended distribution

- Given an instantaneous code $c : S \to \{0,1\}^*$, define a $p : S \to [0,1]$ as
$$p(x) = 2^{-|c(x)|}.$$
- p is called the *intended distribution* for the code c.
- to be more precise: P is a probability distribution if c is complete (otherwise it does not sum up to 1, and we need to introduce some normalization factor to turn it into a distribution).

It is easy to see that (if S is finite) c is the optimal code for p; in fact:

$$H(p) = \sum_{s \in S} -p(s) \log p(s) = \sum_{s \in S} 2^{-|c(s)|} |c(s)| = \sum_{s \in S} p(s)|c(s)| = E_p[c].$$

So, in practice, the choice of the code to use will be based on the expected distribution of the data.

Fixed-length coding

- If $S = \{1, 2, \ldots, N\}$, to represent an element of S it is sufficient to use $\lceil \log N \rceil$ bits.

Fixed-length coding

- If $S = \{1, 2, \ldots, N\}$, to represent an element of S it is sufficient to use $\lceil \log N \rceil$ bits.
- The fixed-length representation for S uses exactly that number of bits for every element (and represents x using the standard binary coding of $x - 1$ on $\lceil \log N \rceil$ bits).

Fixed-length coding

- If $S = \{1, 2, \ldots, N\}$, to represent an element of S it is sufficient to use $\lceil \log N \rceil$ bits.
- The fixed-length representation for S uses exactly that number of bits for every element (and represents x using the standard binary coding of $x - 1$ on $\lceil \log N \rceil$ bits).
- Intended distribution:

$$p(x) = 2^{-\lceil \log N \rceil} \quad \text{uniform distribution.}$$

Unary coding

- If $S = \mathbf{N}$, one can represent $x \in S$ writing x zeroes followed by a one.

Unary coding

- If $S = \mathbf{N}$, one can represent $x \in S$ writing x zeroes followed by a one.
- So $l_x = x + 1$, and the intended distribution is

$$p(x) = 2^{-x-1} \quad \text{geometric distribution of ratio } 1/2.$$

Unary coding

- If $S = \mathbf{N}$, one can represent $x \in S$ writing x zeroes followed by a one.
- So $l_x = x + 1$, and the intended distribution is

$$p(x) = 2^{-x-1} \quad \text{geometric distribution of ratio } 1/2.$$

0	1
1	01
2	001
3	0001
4	00001

A more general viewpoint

Unary coding can be seen as a special case of a more general kind of coding for **N**. Suppose you group **N** into *slots*: every slot is made by consecutive integers; let

$$V = \langle s_1, s_2, s_3, \dots \rangle$$

be the slot sizes (in the unary case $s_1 = s_2 = \cdots = 1$).

A more general viewpoint

Unary coding can be seen as a special case of a more general kind of coding for **N**. Suppose you group **N** into *slots*: every slot is made by consecutive integers; let

$$V = \langle s_1, s_2, s_3, \dots \rangle$$

be the slot sizes (in the unary case $s_1 = s_2 = \cdots = 1$).
Then, to represent $x \in \mathbf{N}$ one can

A more general viewpoint

Unary coding can be seen as a special case of a more general kind of coding for **N**. Suppose you group **N** into *slots*: every slot is made by consecutive integers; let

$$V = \langle s_1, s_2, s_3, \ldots \rangle$$

be the slot sizes (in the unary case $s_1 = s_2 = \cdots = 1$).
Then, to represent $x \in \mathbf{N}$ one can

- encode *in unary* the index i of the slot containing x;

A more general viewpoint

Unary coding can be seen as a special case of a more general kind of coding for **N**. Suppose you group **N** into *slots*: every slot is made by consecutive integers; let

$$V = \langle s_1, s_2, s_3, \dots \rangle$$

be the slot sizes (in the unary case $s_1 = s_2 = \cdots = 1$).
Then, to represent $x \in \mathbf{N}$ one can

- encode *in unary* the index i of the slot containing x;
- encode *in binary* the offset of x within its slot (using $\lceil \log s_i \rceil$ bits).

Golomb coding

Golomb coding with modulus b is obtained choosing

$$V = \langle b, b, b, \ldots \rangle.$$

To represent $x \in \mathbf{N}$ you need to specify the slot where x falls (that is, $\lfloor x/b \rfloor$) in unary, and then represent the offset using $\lceil \log b \rceil$ bits (or $\lfloor \log b \rfloor$ bits).

Golomb coding

Golomb coding with modulus b is obtained choosing

$$V = \langle b, b, b, \dots \rangle.$$

To represent $x \in \mathbf{N}$ you need to specify the slot where x falls (that is, $\lfloor x/b \rfloor$) in unary, and then represent the offset using $\lceil \log b \rceil$ bits (or $\lfloor \log b \rfloor$ bits).
So

$$l_x = \left\lfloor \frac{x}{b} \right\rfloor + \lceil \log b \rceil.$$

The intended distribution is

$$p(x) = 2^{-l_x} \propto (2^{1/b})^{-x} \quad \text{geometric distribution of ratio } 1/\sqrt[b]{2}.$$

More precisely...

A finer analysis shows that Golomb coding is optimal (=Huffman) for a geometric distribution of ratio p, provided that b is chosen as

$$b = \left\lceil \frac{\log(2-p)}{-\log(1-p)} \right\rceil .$$

More precisely...

A finer analysis shows that Golomb coding is optimal (=Huffman) for a geometric distribution of ratio p, provided that b is chosen as

$$b = \left\lceil \frac{\log(2-p)}{-\log(1-p)} \right\rceil .$$

0	**1**0
1	**1**10
2	**1**11
3	**01**0
4	**01**10
5	**01**11
6	**001**0

Elias' γ

Elias' γ *coding* of $x \in \mathbf{N}^+$ is obtained by representing x in binary preceded by a unary representation of its length (minus one).

Elias' γ

Elias' γ *coding* of $x \in \mathbf{N}^+$ is obtained by representing x in binary preceded by a unary representation of its length (minus one). More precisely, to represent x we write in unary $\lfloor \log x \rfloor$ and then in binary $x - 2^{\lceil \log x \rceil}$ (on $\lfloor \log x \rfloor$ bits).

Elias' γ

Elias' γ *coding* of $x \in \mathbf{N}^+$ is obtained by representing x in binary preceded by a unary representation of its length (minus one). More precisely, to represent x we write in unary $\lfloor \log x \rfloor$ and then in binary $x - 2^{\lceil \log x \rceil}$ (on $\lfloor \log x \rfloor$ bits). So

$$l_x = 1 + 2\lfloor \log x \rfloor \quad \Longrightarrow \; p(x) \propto \frac{1}{2x^2}(\text{Zipf of exponent } 2)$$

Elias' γ

Elias' γ *coding* of $x \in \mathbf{N}^+$ is obtained by representing x in binary preceded by a unary representation of its length (minus one).
More precisely, to represent x we write in unary $\lfloor \log x \rfloor$ and then in binary $x - 2^{\lceil \log x \rceil}$ (on $\lfloor \log x \rfloor$ bits). So

$$l_x = 1 + 2\lfloor \log x \rfloor \quad \implies p(x) \propto \frac{1}{2x^2} \text{(Zipf of exponent 2)}$$

1	**1**
2	**01**0
3	**01**1
4	**001**00
5	**001**01

Universal codes

Universal codes

- Given an instantaneous code c for the integers, we say that it is *universal* iff $E_p[c]/H[p]$ is bounded above by a constant for every non-increasing distribution p.

Universal codes

- Given an instantaneous code c for the integers, we say that it is *universal* iff $E_p[c]/H[p]$ is bounded above by a constant for every non-increasing distribution p.
- In other words, a universal code is one that does not loose more than a constant factor with respect to the optimal code *independently from the distribution* (provided that it is non-increasing).

Universal codes

- Given an instantaneous code c for the integers, we say that it is *universal* iff $E_p[c]/H[p]$ is bounded above by a constant for every non-increasing distribution p.
- In other words, a universal code is one that does not loose more than a constant factor with respect to the optimal code *independently from the distribution* (provided that it is non-increasing).
- Elias' γ is the first example we meet of a universal code!

Elias' δ

Elias' δ *coding* of $x \in \mathbf{N}^+$ is obtained by representing x in binary preceded by a representation of its length in γ. [Also δ is universal!]

Elias' δ

Elias' δ *coding* of $x \in \mathbf{N}^+$ is obtained by representing x in binary preceded by a representation of its length in γ. [Also δ is universal!] So

$$l_x = 1 + 2\lfloor \log \log x \rfloor + \lfloor \log x \rfloor \qquad \Longrightarrow \; p(x) \propto \frac{1}{2x(\log x)^2}$$

Elias' δ

Elias' δ *coding* of $x \in \mathbf{N}^+$ is obtained by representing x in binary preceded by a representation of its length in γ. [Also δ is universal!] So

$$l_x = 1 + 2\lfloor \log \log x \rfloor + \lfloor \log x \rfloor \qquad \Longrightarrow \quad p(x) \propto \frac{1}{2x(\log x)^2}$$

1	**1**
2	**010**0
3	**010**1
4	**011**00
5	**011**01
6	**00100**000
7	**00100**001

An alternative way. . .

. . . to think of γ coding is that x is represented using its usual binary representation (except for the initial "1", which is omitted), with every bit "coming with" a continuation bit, that tells whether the representation continues or whether it stops there.
For example (up to bit permutation) γ coding of 724 (in binary: 1011010100) is

0	1	1	1	1	1	0	1	1	1	0	1	1	1	0	1	0	0

k-bit-variable coding

What happens if we group digits k by k?

0 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 0 0

0 0 1 1 1 1 0 1 1 0 1 1 0 0 0

0 1 1 1 0 1 0 1 1 0 0 0

1 1 0 1 1 0 1 0 0 0

k-bit-variable coding (cont'd)

For x, we use $\lceil \log(x)/k \rceil$ bits for the unary part, and the same number of bits multiplied by k for the binary part.

k-bit-variable coding (cont'd)

For x, we use $\lceil \log(x)/k \rceil$ bits for the unary part, and the same number of bits multiplied by k for the binary part.
So

$$l_x = (k{+}1)(\lceil \log(x)/k \rceil) \quad \Longrightarrow \; p(x) \propto x^{-(k+1)/k} (\text{Zipf } (k+1)/k)$$

k-bit-variable coding (cont'd)

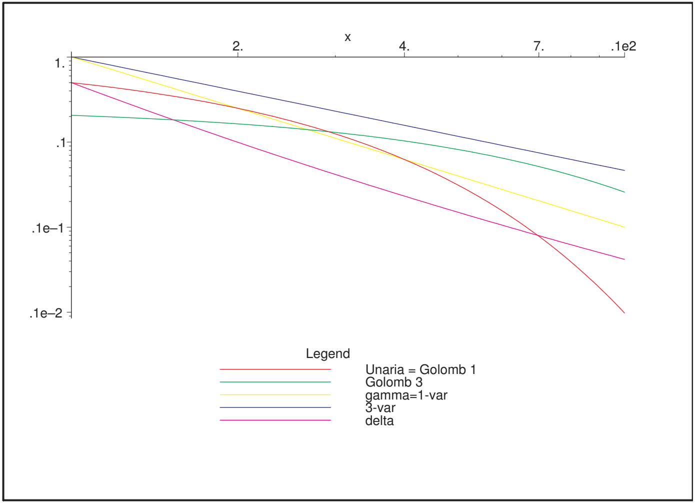
For x, we use $\lceil \log(x)/k \rceil$ bits for the unary part, and the same number of bits multiplied by k for the binary part.
So

$$l_x = (k+1)(\lceil \log(x)/k \rceil) \quad \Longrightarrow \quad p(x) \propto x^{-(k+1)/k} (\text{Zipf } (k+1)/k)$$

A more efficient variant: the ζ_k codes (for Zipf $1 \to 2$).

	$\gamma = \zeta_1$	ζ_2	ζ_3	ζ_4
1	1	10	100	1000
2	010	110	1010	10010
3	011	111	1011	10011
4	00100	01000	1100	10100
5	00101	01001	1101	10101
6	00110	01010	1110	10110
7	00111	01011	1111	10111
8	0001000	011000	0100000	11000

Comparing codings



Graph compression with instantaneous codes: BVGraph

Coding techniques. . .

. . . alone do not improve on compression: we have first to guarantee that the data we represent have a distribution close to the intended one (depending on the coding we are going to use). In particular, they have to enjoy a monotonic distribution (smaller values are more probable than larger ones).

Coding techniques. . .

. . . alone do not improve on compression: we have first to guarantee that the data we represent have a distribution close to the intended one (depending on the coding we are going to use). In particular, they have to enjoy a monotonic distribution (smaller values are more probable than larger ones).

- BTW: some codings (e.g., Elias γ and δ) are universal: for whatever monotonic distribution, they guarantee an expected length that is only within a constant factor of the optimal one.

Coding techniques. . .

. . . alone do not improve on compression: we have first to guarantee that the data we represent have a distribution close to the intended one (depending on the coding we are going to use). In particular, they have to enjoy a monotonic distribution (smaller values are more probable than larger ones).

- BTW: some codings (e.g., Elias γ and δ) are universal: for whatever monotonic distribution, they guarantee an expected length that is only within a constant factor of the optimal one.
- Degrees are often distributed like a Zipf of exponent ≈ 2.7: they can be safely encoded using γ.

Coding techniques...

...alone do not improve on compression: we have first to guarantee that the data we represent have a distribution close to the intended one (depending on the coding we are going to use). In particular, they have to enjoy a monotonic distribution (smaller values are more probable than larger ones).

- BTW: some codings (e.g., Elias γ and δ) are universal: for whatever monotonic distribution, they guarantee an expected length that is only within a constant factor of the optimal one.
- Degrees are often distributed like a Zipf of exponent ≈ 2.7: they can be safely encoded using γ.
- What about successors? Let us assume that successors of x are $y_1, \ldots, y_k$: how should we encode $y_1, \ldots, y_k$?

Locality

In general, we cannot say much about their distribution, unless we make some assumption on the way in which nodes are numbered.

Locality

In general, we cannot say much about their distribution, unless we make some assumption on the way in which nodes are numbered. **The following considerations hold true for web graphs!**

Locality

In general, we cannot say much about their distribution, unless we make some assumption on the way in which nodes are numbered. **The following considerations hold true for web graphs!**

- Many hypertextual links contained in a web page are *navigational* ("home", "next", "up"...). If we compare the URL they refer to with that of the page containing them, they share a long common prefix. This property is known as *locality* and it was first observed by the authors of the Connectivity Server.

Locality

In general, we cannot say much about their distribution, unless we make some assumption on the way in which nodes are numbered. **The following considerations hold true for web graphs!**

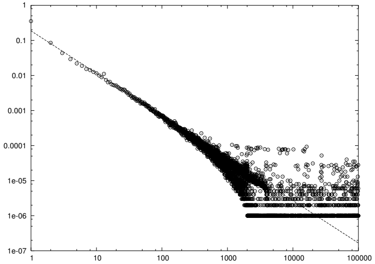
- Many hypertextual links contained in a web page are *navigational* ("home", "next", "up"...). If we compare the URL they refer to with that of the page containing them, they share a long common prefix. This property is known as *locality* and it was first observed by the authors of the Connectivity Server.
- To exploit this property, assume that URLs are ordered lexicographically (that is, node 0 is the first URL in lexicographic order, etc.). Then, if $x \to y$ is an arc, most of the times $|x - y|$ will be "small".

Exploiting locality

If x has successors $y_1 < y_2 < \cdots < y_k$, we represent its successor list though the gaps (*differentiation*):

$$y_1 - x, y_2 - y_1 - 1, \ldots, y_k - y_{k-1} - 1$$

(only the first value can be negative: wa make it into a natural number...). How are such differences distributed?



Zipf with exponent 1.2 $\implies$ we use ζ_3.

Similarity

URLs close to each other (in lexicographic order) have similar successor sets: this fact (known as *similarity*) was exploited for the first time in the Link database.

Similarity

URLs close to each other (in lexicographic order) have similar successor sets: this fact (known as *similarity*) was exploited for the first time in the Link database.
We may encode the successor list of x as follows:

Similarity

URLs close to each other (in lexicographic order) have similar successor sets: this fact (known as *similarity*) was exploited for the first time in the Link database.

We may encode the successor list of x as follows:

- we write the differences with respect to the successor list of some previous node $x - r$ (called the *reference node*)

Similarity

URLs close to each other (in lexicographic order) have similar successor sets: this fact (known as *similarity*) was exploited for the first time in the Link database.

We may encode the successor list of x as follows:

- we write the differences with respect to the successor list of some previous node $x - r$ (called the *reference node*)
- we explicitly encode (as before) only the successors of x that were not successors of $x - r$.

Similarity (cont'd)

More explicitly, the successor list of x is encoded as (*referencing*):

- an intger r (reference): if $r > 0$, the list is described by difference with respect to the successor list of $x - r$; in this case, we write a bitvector (of length equal to $d^+(x - r)$) discriminating the elements in $N^+(x - r) \cap N^+(x)$ from the ones in $N^+(x - r) \setminus N^+(x)$

More explicitly, the successor list of x is encoded as (*referencing*):

- an intger r (reference): if $r > 0$, the list is described by difference with respect to the successor list of $x - r$; in this case, we write a bitvector (of length equal to $d^+(x - r)$) discriminating the elements in $N^+(x - r) \cap N^+(x)$ from the ones in $N^+(x - r) \setminus N^+(x)$
- an explicit list of *extra nodes*, containing the elements of $N^+(x) \setminus N^+(x - r)$ (or the whole $N^+(x)$, if $r = 0$), encoded as explained before.

Referencing example

Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Blocks (*differential compression*)

Instead of using a bitvector, we use run-length encoding, telling the length of successive runs (blocks) of "0" and "1":

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Blocks (*differential compression*)

Instead of using a bitvector, we use run-length encoding, telling the length of successive runs (blocks) of “0” and “1”:

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Node	Outd.	Ref.	# blocks	Copy blocks	Extra nodes
...	...	...	...	...	...
15	11	0			13, 15, 16, 17, 18, 19, 23, ...
16	10	1	7	0, 0, 2, 1, 1, 0, 0	22, 316, ...
17	0				
18	5	3	1	4	50
...	...	...	...	...	...

Consecutivity

Among the extra nodes, many happen to sport the *consecutivity* property: they appear in clusters of consecutive integers. This phenomenon, observed empirically, have some possible explanations:

Consecutivity

Among the extra nodes, many happen to sport the *consecutivity* property: they appear in clusters of consecutive integers. This phenomenon, observed empirically, have some possible explanations:

- most pages contain groups of navigational links that correspond to a certain hierarchical level of the website, and are often consecutive to one another;

Consecutivity

Among the extra nodes, many happen to sport the *consecutivity* property: they appear in clusters of consecutive integers. This phenomenon, observed empirically, have some possible explanations:

- most pages contain groups of navigational links that correspond to a certain hierarchical level of the website, and are often consecutive to one another;
- in the transpose graph, moreover, consecutivity is the dual of similarity with reference 1: when there is a cluster of consecutive pages with many similar links, in the transpose there are intervals of consecutive outgoing links.

Consecutivity (cont'd)

To exploit consecutivity, we use a special representation for the extra node list called *intervalization*, that is:

- sufficiently long (say $\geq T$) intervals of consecutive integers are represented by their left extreme and their length minus T;
- other extra nodes, if any, are called *residual nodes* and are represented alone.

Intervalization example

Node	Outd.	Ref.	# blocks	Copy blocks	Extra nodes
...	...	...	...	...	...
15	11	0			13, 15, 16, 17, 18, 19, 23, ...
16	10	1	7	0, 0, 2, 1, 1, 0, 0	22, 316, ...
17	0				
18	5	3	1	4	50
...	...	...	...	...	...

Intervalization example

Node	Outd.	Ref.	# blocks	Copy blocks	Extra nodes
...	...	...	...	...	...
15	11	0			13, 15, 16, 17, 18, 19, 23, ...
16	10	1	7	0, 0, 2, 1, 1, 0, 0	22, 316, ...
17	0				
18	5	3	1	4	50
...	...	...	...	...	...

Node	Outd.	Ref.	# bl.	Copy bl.s	# int.	Lft extr.	Lth	Residuals
...	...	...	...	...	...	...	...	...
15	11	0			2	15,...	4,...	13, 23 ...
16	10	1	7	0, 0, ...	1	316	1	22, 3041
17	0							
18	5	3	1	4	0			50
...	...	...	...	...	...	...	...	...

Reference window

When the reference node is chosen, how far back in the "past" are we allowed to go?

Reference window

When the reference node is chosen, how far back in the "past" are we allowed to go? We need to keep track of a window of the last W successor lists. The choice of W is critical:

Reference window

When the reference node is chosen, how far back in the "past" are we allowed to go? We need to keep track of a window of the last W successor lists. The choice of W is critical:

- a large W guarantees better compression, but increases compression time and space

Reference window

When the reference node is chosen, how far back in the "past" are we allowed to go? We need to keep track of a window of the last W successor lists. The choice of W is critical:

- a large W guarantees better compression, but increases compression time and space
- after $W = 7$ there is no significant improvement in compression.

Reference window

When the reference node is chosen, how far back in the "past" are we allowed to go? We need to keep track of a window of the last W successor lists. The choice of W is critical:

- a large W guarantees better compression, but increases compression time and space
- after $W = 7$ there is no significant improvement in compression.

The choice of W does not impact on decompression time.

Reference chain length

Referencing involves recursion: to decode the successor list of x, we need first to decompress the successor list of $x - r$, etc. This chain is called the *reference chain* of x: decompression speed depends on the length of such chains.

Reference chain length

Referencing involves recursion: to decode the successor list of x, we need first to decompress the successor list of $x - r$, etc. This chain is called the *reference chain* of x: decompression speed depends on the length of such chains.
During compression, it is possible to limit their length keeping into account of how long is the reference chain for every node in the window and avoiding to use nodes whose reference chain is already of a given maximum length R.

Reference chain length

Referencing involves recursion: to decode the successor list of x, we need first to decompress the successor list of $x - r$, etc. This chain is called the *reference chain* of x: decompression speed depends on the length of such chains.

During compression, it is possible to limit their length keeping into account of how long is the reference chain for every node in the window and avoiding to use nodes whose reference chain is already of a given maximum length R.

The choice of R influences the compression ratio (with $R = \infty$ giving the best possible compression) but also on decompression speed ($R = \infty$ may produce access time that can be two orders of magnitude larger than $R = 1$ — it may even produce stack overflows).

BVGraph for general graphs: LLPA

The basic property we have been exploiting so far is that *nodes are numbered according to the lexicographic ordering of URLs*. Is it possible to adapt / extend this idea to non-web graphs, e.g., to social networks?

From web graphs to complex networks

The basic property we have been exploiting so far is that *nodes are numbered according to the lexicographic ordering of URLs*. Is it possible to adapt / extend this idea to non-web graphs, e.g., to social networks?

- What we want is an ordering of the nodes that is compression friendly

From web graphs to complex networks

The basic property we have been exploiting so far is that *nodes are numbered according to the lexicographic ordering of URLs*. Is it possible to adapt / extend this idea to non-web graphs, e.g., to social networks?

- What we want is an ordering of the nodes that is compression friendly
- In particular, we want that most arcs are between nodes that are very close (as numbers) to each other.

Orderings and communities

Social networks have no natural ordering such as "lexicographic by URL". However many statistics suggest that social networks are clustered.

Orderings and communities

Social networks have no natural ordering such as “lexicographic by URL”. However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.

Orderings and communities

Social networks have no natural ordering such as "lexicographic by URL". However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.

Constraints:

Orderings and communities

Social networks have no natural ordering such as "lexicographic by URL". However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.

Constraints:

1. very few clustering techniques scale up to very large graphs

Orderings and communities

Social networks have no natural ordering such as “lexicographic by URL”. However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.

Constraints:

1. very few clustering techniques scale up to very large graphs
2. we do not posses any prior information on the number of clusters

Orderings and communities

Social networks have no natural ordering such as "lexicographic by URL". However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.

Constraints:

1. very few clustering techniques scale up to very large graphs
2. we do not posses any prior information on the number of clusters
3. cluster sizes are going to be very unbalanced

Orderings and communities (cont'd)

Orderings and communities (cont'd)

- You can obtain an ordering from a clustering just sorting by cluster label

Orderings and communities (cont'd)

- You can obtain an ordering from a clustering just sorting by cluster label
- Different clustering algorithms yield different and incomparable orderings

Orderings and communities (cont'd)

- You can obtain an ordering from a clustering just sorting by cluster label
- Different clustering algorithms yield different and incomparable orderings
- Main idea:

Orderings and communities (cont'd)

- You can obtain an ordering from a clustering just sorting by cluster label
- Different clustering algorithms yield different and incomparable orderings
- Main idea:
  - Run a clustering algorithm A

Orderings and communities (cont'd)

- You can obtain an ordering from a clustering just sorting by cluster label
- Different clustering algorithms yield different and incomparable orderings
- Main idea:
  - Run a clustering algorithm A
  - Renumber nodes sorting by A's labels, breaking ties using the node numbers (i.e., sort stably by A's labels)

Orderings and communities (cont'd)

- You can obtain an ordering from a clustering just sorting by cluster label
- Different clustering algorithms yield different and incomparable orderings
- Main idea:
  - Run a clustering algorithm A
  - Renumber nodes sorting by A's labels, breaking ties using the node numbers (i.e., sort stably by A's labels)
  - Iterate with another clustering algorithm

Label Propagation Algorithm (LPA)

LPA are a class of clustering algorithm that work as follows:

Label Propagation Algorithm (LPA)

LPA are a class of clustering algorithm that work as follows:

- Every node adopts the label that is most common among its neighbors. . .

Label Propagation Algorithm (LPA)

LPA are a class of clustering algorithm that work as follows:

- Every node adopts the label that is most common among its neighbors...
- ...with an adjustment depending on the overall popularity of the label

Label Propagation Algoritm (LPA)

Require: G a graph, γ a density parameter

$\pi \leftarrow$ a random permutation of G's nodes
for all x: $\lambda(x) \leftarrow x$, $v(x) \leftarrow 1$
while (some stopping criterion) **do**
    for $i = 0, 1, \ldots, n-1$ **do**
        for every label ℓ, $k_\ell \leftarrow |\lambda^{-1}(\ell) \cap N_G(\pi(i))|$
        $\hat{\ell} \leftarrow \text{argmax}_\ell[k_\ell - \gamma(v(\ell) - k_\ell)]$
        decrement $v(\lambda(\pi(i)))$
        $\lambda(\pi(i)) \leftarrow \hat{\ell}$
        increment $v(\lambda(\pi(i)))$
    end for
end while

Here $v(\ell)$ is the number of nodes currently labelled by ℓ, so $v(\ell) - k_\ell$ is the popularity of label ℓ outside of the current neighborhood.

Layered Label Propagation Algoritm (LLPA)

Layered Label Propagation Algoritm (LLPA)

- Repeatedly run LPA with different values of γ

Layered Label Propagation Algoritm (LLPA)

- Repeatedly run LPA with different values of γ
- Renumber nodes sorting by stably by the new labels

Name	LLP		BFS	Shingle		Natural		Random	
Amazon	9.16	(-30%)	12.96	14.43	(+11%)	16.92	(+30%)	23.62	(+82%)
DBLP	6.88	(-23%)	8.91	11.42	(+28%)	11.36	(+27%)	22.07	(+147%)
Enron	6.51	(-24%)	8.54	9.87	(+15%)	13.43	(+57%)	14.02	(+64%)
Hollywood	5.14	(-35%)	7.81	6.72	(-14%)	15.20	(+94%)	16.23	(+107%)
LiveJournal	10.90	(-28%)	15.1	15.77	(+4%)	14.35	(-5%)	23.50	(+55%)
Flickr	8.89	(-22%)	11.26	10.22	(-10%)	13.87	(+23%)	14.49	(+28%)
indochina (hosts)	5.53	(-17%)	6.63	7.16	(+7%)	9.26	(+39%)	10.59	(+59%)
uk (hosts)	6.26	(-18%)	7.62	8.12	(+6%)	10.81	(+41%)	15.58	(+104%)
eu	3.90	(-21%)	4.93	6.86	(+39%)	5.24	(+6%)	19.89	(+303%)
in	2.46	(-30%)	3.51	4.79	(+36%)	2.99	(-15%)	21.15	(+502%)
indochina	1.71	(-26%)	2.31	3.59	(+55%)	2.19	(-6%)	21.46	(+829%)
it	2.10	(-28%)	2.89	4.39	(+51%)	2.83	(-3%)	26.40	(+813%)
uk	1.91	(-33%)	2.84	4.09	(+44%)	2.75	(-4%)	27.55	(+870%)
altavista-nd	5.22	(-11%)	5.85	8.12	(+38%)	8.37	(+43%)	34.76	(+494%)

Graph compression with Elias-Fano: EFGraph

Elias-Fano representation

Elias-Fano representation

- Elias proposed in 1975 a *general* representation for monotone sequences, later discussed by Fano

Elias-Fano representation

- Elias proposed in 1975 a *general* representation for monotone sequences, later discussed by Fano
- In 2012 Vigna proposed to use it for inverted indices, and in particular for storing successor lists in graph compression...

Elias-Fano representation

- Elias proposed in 1975 a *general* representation for monotone sequences, later discussed by Fano
- In 2012 Vigna proposed to use it for inverted indices, and in particular for storing successor lists in graph compression. . .
- A very general technique!

Idea

Given a non-decreasing sequence:

$$0 \leq x_1, \ldots, x_d < n$$

(e.g., the successors of a node of degree d in a graph of n nodes):

Idea

Given a non-decreasing sequence:

$$0 \leq x_1, \ldots, x_d < n$$

(e.g., the successors of a node of degree d in a graph of n nodes):

- let $\ell = \lfloor \log(n/d) \rfloor$

Idea

Given a non-decreasing sequence:

$$0 \leq x_1, \ldots, x_d < n$$

(e.g., the successors of a node of degree d in a graph of n nodes):

- let $\ell = \lfloor \log(n/d) \rfloor$
- write the ℓ lower bits of x_i explicitly

Idea

Given a non-decreasing sequence:

$$0 \leq x_1, \ldots, x_d < n$$

(e.g., the successors of a node of degree d in a graph of n nodes):

- let $\ell = \lfloor \log(n/d) \rfloor$
- write the ℓ lower bits of x_i explicitly
- write the upper bits as gaps, written in unary

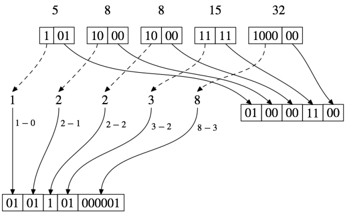
Idea

Given a non-decreasing sequence:

$$0 \leq x_1, \ldots, x_d < n$$

(e.g., the successors of a node of degree d in a graph of n nodes):

- let $\ell = \lfloor \log(n/d) \rfloor$
- write the ℓ lower bits of x_i explicitly
- write the upper bits as gaps, written in unary



5, 8, 8, 15, 32 ≤ n = 36, ℓ = 2

Idea

So $x_1 = y_1 \cdot 2^\ell + r_1$, ..., $x_d = y_d \cdot 2^\ell + r_d$, with r_i written using ℓ bits each, and $g_1 = y_1 - 0, g_2 = y_2 - y_1, \ldots, g_d = y_d - y_{d-1}$ written in unary:

Idea

So $x_1 = y_1 \cdot 2^\ell + r_1$, ..., $x_d = y_d \cdot 2^\ell + r_d$, with r_i written using ℓ bits each, and $g_1 = y_1 - 0, g_2 = y_2 - y_1, \ldots, g_d = y_d - y_{d-1}$ written in unary:

- writing the g_i's requires d stopping bits...

Idea

So $x_1 = y_1 \cdot 2^\ell + r_1$, ..., $x_d = y_d \cdot 2^\ell + r_d$, with r_i written using ℓ bits each, and $g_1 = y_1 - 0, g_2 = y_2 - y_1, \ldots, g_d = y_d - y_{d-1}$ written in unary:

- writing the g_i's requires d stopping bits...
- ...plus $g_1 + \cdots + g_d$ bits

Idea

So $x_1 = y_1 \cdot 2^\ell + r_1$, ..., $x_d = y_d \cdot 2^\ell + r_d$, with r_i written using ℓ bits each, and $g_1 = y_1 - 0, g_2 = y_2 - y_1, \ldots, g_d = y_d - y_{d-1}$ written in unary:

- writing the g_i's requires d stopping bits...
- ...plus $g_1 + \cdots + g_d$ bits
- $g_1 + \cdots + g_d = y_d \leq n/2^\ell \leq n/2^{\log(n/d)-1} = 2d$ bits.

Idea

So $x_1 = y_1 \cdot 2^\ell + r_1, \ldots, x_d = y_d \cdot 2^\ell + r_d$, with r_i written using ℓ bits each, and $g_1 = y_1 - 0, g_2 = y_2 - y_1, \ldots, g_d = y_d - y_{d-1}$ written in unary:

- writing the g_i's requires d stopping bits...
- ... plus $g_1 + \cdots + g_d$ bits
- $g_1 + \cdots + g_d = y_d \leq n/2^\ell \leq n/2^{\log(n/d)-1} = 2d$ bits.

All in all we need $2 + \lceil \log(n/d) \rceil$ bits per element.

Idea

So $x_1 = y_1 \cdot 2^\ell + r_1, \ldots, x_d = y_d \cdot 2^\ell + r_d$, with r_i written using ℓ bits each, and $g_1 = y_1 - 0, g_2 = y_2 - y_1, \ldots, g_d = y_d - y_{d-1}$ written in unary:

- writing the g_i's requires d stopping bits...
- ...plus $g_1 + \cdots + g_d$ bits
- $g_1 + \cdots + g_d = y_d \le n/2^\ell \le n/2^{\log(n/d)-1} = 2d$ bits.

All in all we need $2 + \lceil \log(n/d) \rceil$ bits per element.

The representation is almost optimal (Elias proves that it is $< .5$ bit away from the information-theoretic lower bound).